

# Deep learning workshop

## Summary and homework

### Libraries

There are three main libraries you always want to import:

```
daniel@pc:~$ python3
Python 3.5.2 (default, Oct  8 2019, 13:06:37)
[GCC 5.4.0 20160609] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>> import torch
>>> import torch.nn as nn
>>> import torch.nn.functional as F
```

Here torch is the main mathematical library, in theory you can do everything with this module alone. In practice however, you will use mostly the other imports (torch.nn & torch.nn.functional). They are there to make life easy, they contain shortcuts to quickly build deep learning models.

### Tensors

Tensors are multidimensional matrices. They are used for numeric computations. Make sure that your input and output are pytorch tensors.

To create a tensor from a **list** or a **numpy array** :

```
variable_name = torch.Tensor([1,2,-3.6])
variable_name = torch.Tensor(your_numpy_array)
```

You can also create tensors using these common used methods:

torch.zeros(size) → creates a tensor filled with zeros with the given shape

torch.ones(size) → creates a tensor filled with ones with the given shape

torch.rand(size) → creates a tensor filled with numbers sampled from an uniform random distribution between 0 and 1

torch.randn(size) → creates a tensor filled with numbers sampled from a normal distribution

torch.eye(n) →

```
>>> torch.eye(4)
tensor([[1., 0., 0., 0.],
        [0., 1., 0., 0.],
        [0., 0., 1., 0.],
        [0., 0., 0., 1.]])
```

A tensor can have multiple dimensions. Since it is hard to imagine anything beyond three dimensions, below you can see how a four dimensional tensor looks like:

```

>>> torch.rand(2,2,2,3)
tensor([[[[0.5724, 0.1078, 0.8437],
          [0.5490, 0.7806, 0.4171]],
        [[0.9159, 0.3229, 0.6921],
          [0.5713, 0.3567, 0.5608]]],
       [[[[0.6300, 0.1676, 0.5844],
          [0.2391, 0.1071, 0.3551]],
        [[0.5107, 0.3989, 0.8393],
          [0.8733, 0.5533, 0.3701]]]])

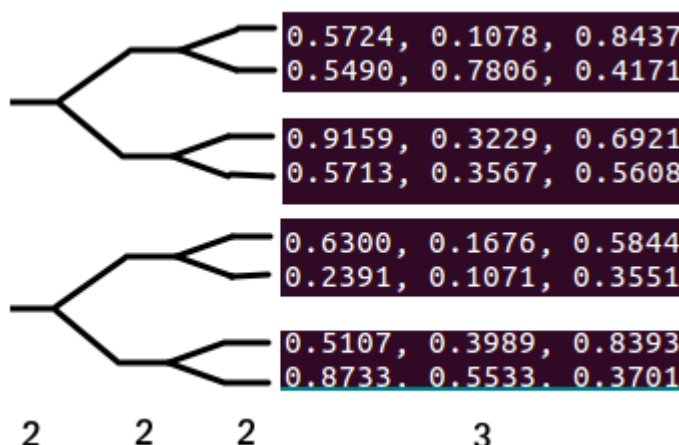
>>> torch.rand(2,2,2,3)
tensor([[[[0.5724, 0.1078, 0.8437],
          [0.5490, 0.7806, 0.4171]],
        [[0.9159, 0.3229, 0.6921],
          [0.5713, 0.3567, 0.5608]]],
       [[[[0.6300, 0.1676, 0.5844],
          [0.2391, 0.1071, 0.3551]],
        [[0.5107, 0.3989, 0.8393],
          [0.8733, 0.5533, 0.3701]]]])

>>> torch.rand(2,2,2,3)
tensor([[[[0.5724, 0.1078, 0.8437],
          [0.5490, 0.7806, 0.4171]],
        [[0.9159, 0.3229, 0.6921],
          [0.5713, 0.3567, 0.5608]]],
       [[[[0.6300, 0.1676, 0.5844],
          [0.2391, 0.1071, 0.3551]],
        [[0.5107, 0.3989, 0.8393],
          [0.8733, 0.5533, 0.3701]]]])

>>> torch.rand(2,2,2,3)
tensor([[[[0.5724, 0.1078, 0.8437],
          [0.5490, 0.7806, 0.4171]],
        [[0.9159, 0.3229, 0.6921],
          [0.5713, 0.3567, 0.5608]]],
       [[[[0.6300, 0.1676, 0.5844],
          [0.2391, 0.1071, 0.3551]],
        [[0.5107, 0.3989, 0.8393],
          [0.8733, 0.5533, 0.3701]]]])

```

If you view the dimensions more like the branching points of an expending tree structure, it is not hard to visualize multiple dimensional tensors. Luckily most deep learning applications won't go beyond four dimensions, DeepRank is a rare exception though.



Retrieving the right-top corner value (0.8437) can be done by: `tensor[0, 0, 0, 2]`

Retrieving the bottom row (0.8733, 0.5533, 0.3701) can be done by: `tensor[1, 1, 1, :]`

The order of dimensions (or axis as it called often) of your input is arbitrary of course. Take, for example, images/photos. Here most libraries define the dimensions as: (y, x, RGB). A photo of 100 pixels in width and 50 pixels in height would have the shape: (50, 100, 3). Pytorch however, requires a different format: (sample, RGB, y, x). This because it can process multiple samples/photos at the same time. Therefore the first dimension is reserved for samples and the second for features. For fully connected layers this is the complete input, for multidimensional data, such as photos, or DeepRank cubes, the other y x z dimensions follow after the first two.

If you want to swap axis (dimensions) use the **permute** function:

```
>>> photo = torch.rand(50,100,3)
>>> photo2 = photo.permute(2,0,1)
>>>
>>> print(photo.shape, '->', photo2.shape)
torch.Size([50, 100, 3]) -> torch.Size([3, 50, 100])
>>>
```

### Common operations on tensors

Given these two tensors:

```
>>> a = torch.rand(5)
>>> b = torch.randn(5)
>>>
>>> a
tensor([0.6171, 0.1505, 0.2631, 0.8784, 0.6518])
>>> b
tensor([ 1.5799, -0.4528,  1.3017, -0.8588,  2.9347])
```

Addition: `new_tensor = a + b`    **or**    `new_tensor = a.add(b)`

Subtraction works the same.

Power: `new_tensor = a**2`    **or**    `new_tensor = a.pow(2)`

Multiplication:

```
>>> a*b
tensor([ 0.9750, -0.0681,  0.3425, -0.7544,  1.9129])
>>>
>>> a.mul(b)
tensor([ 0.9750, -0.0681,  0.3425, -0.7544,  1.9129])
>>>
>>> a * 10
tensor([6.1713, 1.5047, 2.6312, 8.7843, 6.5180])
```

Dot product:

```
>>> inp = torch.rand(10)
>>> W = torch.randn(5,10)
>>> torch.matmul(W, inp)
tensor([-1.2377, -0.1801, -0.8193,  0.5665,  1.0289])
>>>
```

Repeat:

```
>>> r = torch.Tensor(range(5))
>>> r
tensor([0., 1., 2., 3., 4.])
>>> r.repeat(5, 1)
tensor([[0., 1., 2., 3., 4.],
        [0., 1., 2., 3., 4.],
        [0., 1., 2., 3., 4.],
        [0., 1., 2., 3., 4.],
        [0., 1., 2., 3., 4.]])
```

`torch.arange(5)`

View:

```
>>> r = torch.arange(1,10)
>>> r
tensor([1, 2, 3, 4, 5, 6, 7, 8, 9])
>>> r.view(3,3)
tensor([[1, 2, 3],
        [4, 5, 6],
        [7, 8, 9]])
```

You will use this 'view' method a lot.

You can also use it to add a new dimension:

```
>>> r = torch.rand(3,3)
>>> r.shape
torch.Size([3, 3])
>>> r = r.view(1,3,3)
>>> r.shape
torch.Size([1, 3, 3])
```

DeepRank uses this often:  
`r.view(-1)`  
`r.reshape(-1)` is also fine

Or combine dimensions: (for example to make 2d images 1d for logistic regression)

```
>>> r = torch.rand(2,5,5)
>>> r = r.view(2, 25)
>>> r_back = r.view(2,5,5)
```

→ combining is reversible

And of course, all the basic math operation you can also find in libraries such as numpy. Think of:

`torch.log`, `torch.exp`, `torch.sqrt`, `torch.sin` ...

If you want to have a tensor with values to optimize, use the 'requires\_grad=True' parameter:

```
>>> torch.rand(2,2, requires_grad=True)
tensor([[0.8711, 0.7144],
        [0.3464, 0.2729]], requires_grad=True)
```

Then pytorch knows it should memorize all operations done with this tensor so that the gradients can be calculated for each parameter in the tensor.

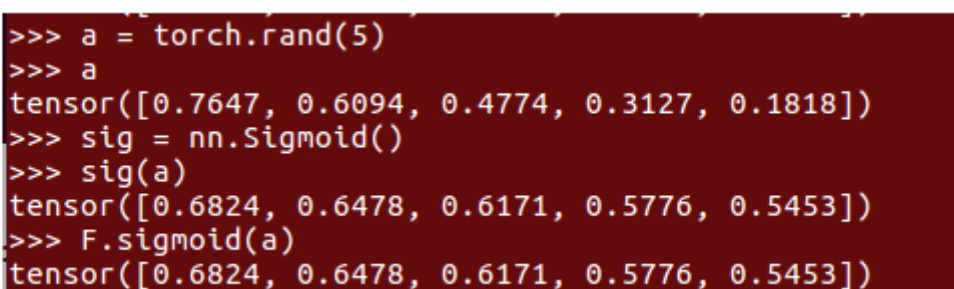
## Torch nn and nn.functional (F)

nn is the **neural network** part of pytorch.

Nn contains mostly the **modules**, and **F (nn.functional)** specific functions.

What is the difference?

Modules are classes that you have to initiate first before using them, while functions can be used directly. Example:



```
>>> a = torch.rand(5)
>>> a
tensor([0.7647, 0.6094, 0.4774, 0.3127, 0.1818])
module — >>> sig = nn.Sigmoid()
>>> sig(a)
tensor([0.6824, 0.6478, 0.6171, 0.5776, 0.5453])
function — >>> F.sigmoid(a)
tensor([0.6824, 0.6478, 0.6171, 0.5776, 0.5453])
```

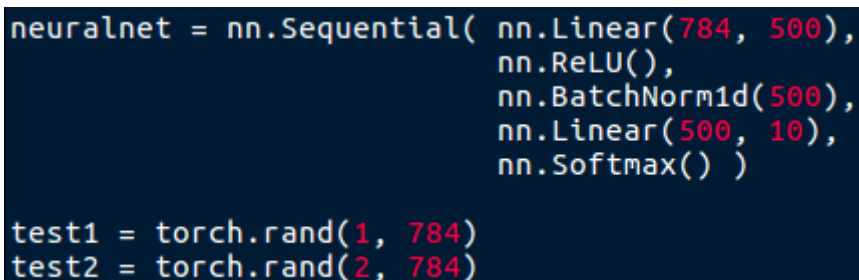
Also modules can contain learnable weights you might want to optimize.

The most common modules are:

- Common activation functions: `nn.Sigmoid()` `nn.ReLU()` `nn.PReLU()` `nn.Tanh()`  
`nn.Softmax(dimension)` `nn.ELU()`
- Regularization: `nn.BatchNorm1d(nmb_neurons)` `nn.Dropout(probability)`  
`nn.BatchNorm2d(nmb_neurons)`
- Layers (weights and biases):  
`nn.Linear(nmb_input_features, nmb_output_features)`  
`nn.Conv1d(nmb_input_features, nmb_output_features, kernel_size)`  
`nn.Conv2d(nmb_input_features, nmb_output_features, kernel_size)`  
(for now we only use `nn.Linear`)

A neural network can be created by combining these modules.

The simplest way is with the `nn.Sequential` module:



```
neuralnet = nn.Sequential( nn.Linear(784, 500),
                           nn.ReLU(),
                           nn.BatchNorm1d(500),
                           nn.Linear(500, 10),
                           nn.Softmax() )

test1 = torch.rand(1, 784)
test2 = torch.rand(2, 784)
```

Here each module is executed one after the other, this is the most efficient way to construct a 'simple' neural network. The network above accepts an input of 784 features and returns a probability distribution over 10 output neurons.

---

#### Homework question 1

If 'test1' is put in the neural network ( `neuralnet(test1)` ), an error occurs. But if test2 is used as input, no error occurs. Why?

---

#### Homework question 2

The images of the MNIST dataset (the handwritten digits) have a shape of 28 x 28 pixels. If we try to feed the neuralnet with 32 samples (with the shape (32, 28, 28)) an error occurs because it expects an input of 32 x 784. How do we adjust the input?

---

You can also create your own modules. Here is an example of an own made exponential activation function:

Must be the same name

Inherits functions and variables from `nn.Module`

```
class MyOwnModule(nn.Module):
    def __init__(self, p1):
        super(MyOwnModule, self).__init__()
        self.p1 = p1
    def forward(self, x):
        return x.pow(self.p1)
```

Here you inherit variables/properties and methods/functions from the standard '`nn.Module`'. The only necessary things to add are the `__init__` function, with your own parameters if needed, and the **forward** function, which is called when you use the module.

---

#### Homework question 3

Create two activation functions just as shown above. The first will be easy, the sigmoid function. The input can be a tensor of any size, and the output should be the sigmoid function applied over all the values. You are not allowed to use for loops fyi.

The second function will be harder, the softmax function. Find out how this function works and implement it like above. The input will be a 2 dimensional matrix. The output will be the same shape. One catch, the probability distributions can be per row or per column. I want to be able to specify this in the `__init__` method.

(make use of the `sum()` method of torch to create the softmax function)

---

## Next step

As shown earlier, the `nn.Sequential` module can be used to create simple neural network architectures. But to create more complicated neural network architectures (like the one Jarek showed), you also have to build them like a module.

An example from the pytorch site (no need to know all the details):

```
class Net(nn.Module):

    def __init__(self):
        super(Net, self).__init__()
        # 1 input image channel, 6 output channels, 3x3 square convolution
        # kernel
        self.conv1 = nn.Conv2d(1, 6, 3)
        self.conv2 = nn.Conv2d(6, 16, 3)
        # an affine operation: y = Wx + b
        self.fc1 = nn.Linear(16 * 6 * 6, 120) # 6*6 from image dimension
        self.fc2 = nn.Linear(120, 84)
        self.fc3 = nn.Linear(84, 10)

    def forward(self, x):
        # Max pooling over a (2, 2) window
        x = F.max_pool2d(F.relu(self.conv1(x)), (2, 2))
        # If the size is a square you can only specify a single number
        x = F.max_pool2d(F.relu(self.conv2(x)), 2)
        x = x.view(-1, self.num_flat_features(x))
        x = F.relu(self.fc1(x))
        x = F.relu(self.fc2(x))
        x = self.fc3(x)
        return x

    def num_flat_features(self, x):
        size = x.size()[1:] # all dimensions except the batch dimension
        num_features = 1
        for s in size:
            num_features *= s
        return num_features

net = Net()
```

Notice that you can use other modules (such as `nn.Linear` and `nn.Conv2d`) within the module itself.



## Loss function and optimizer

The loss function is a description of your objective. The output will be a score (the loss) of how well the neural network performs.

The mean squared error loss function, for example, can be done with:

```
loss = (prediction-true_labels)**2
```

or

```
loss = F.mse_loss(prediction, true_labels)
```

You can add many objectives to the loss function, as long as the final loss is a single number.

```
Loss = F.mse_loss(prediction, true_labels) + (weights)**2 * 0.001
```

Above, for example, L2 regularization is added to the loss (which is called `weight_decay` in deep learning). Notice that it is multiplied by a small number (0.001) so that this part of the loss will not dominate over the main objective we try to solve.

In the near future we are going to use cross entropy and binary cross entropy.

To calculate the gradients for all the learnable parameters, just use:

```
loss.backward()
```

Now the gradients are calculated, but not yet applied to the learnable weights. This can be done with an optimizer:

```
optim = torch.optim.SGD( neuralnet.parameters(), lr=0.001)
```

This is an example of a standard stochastic gradient descent optimizer, which does the operation we saw many times before :  $\text{new\_weights} = \text{old\_weights} - \text{gradients} * \text{learning\_rate}$

The `parameters()` method retrieves the learnable weights from your neural network.

There are many optimizers that can use smart tricks to make optimization much faster, but at the cost that it might not find the same (sometimes worse) solution. The one best to use most of the time is adam:

```
optim = torch.optim.Adam( neuralnetwork.parameters() )
```

I will spare you the details why this optimizer is so effective. Just know this will almost always work, although published deep learning paper researchers still prefer SGD instead.

Once you defined the optimizer and calculated the gradients with the `loss.backward()` function, you can use the optimizer with these commands:



```
optim.step()

optim.zero_grad()
```

The step function applies the gradients to the current weights and the zero\_grad() functions clears the gradients from memory. If the old gradients are not cleared, the new gradients will be added to the old gradients, which can be undesirable.

---

#### Homework question 4.

To summarize how to build and train a neural network can be done with the following pseudocode:

**define a network** → `net = nn.Sequential( nn.Linear....`

**define an optimizer** → `optim = torch.optim.SGD(net.parameters(), lr=0.01)`

**define data** → `data = torch.Tensor(...`

**Define labels** → `labels = torch.Tensor(...`

**Train your network: while not converged:**

`prediction = net ( data )`

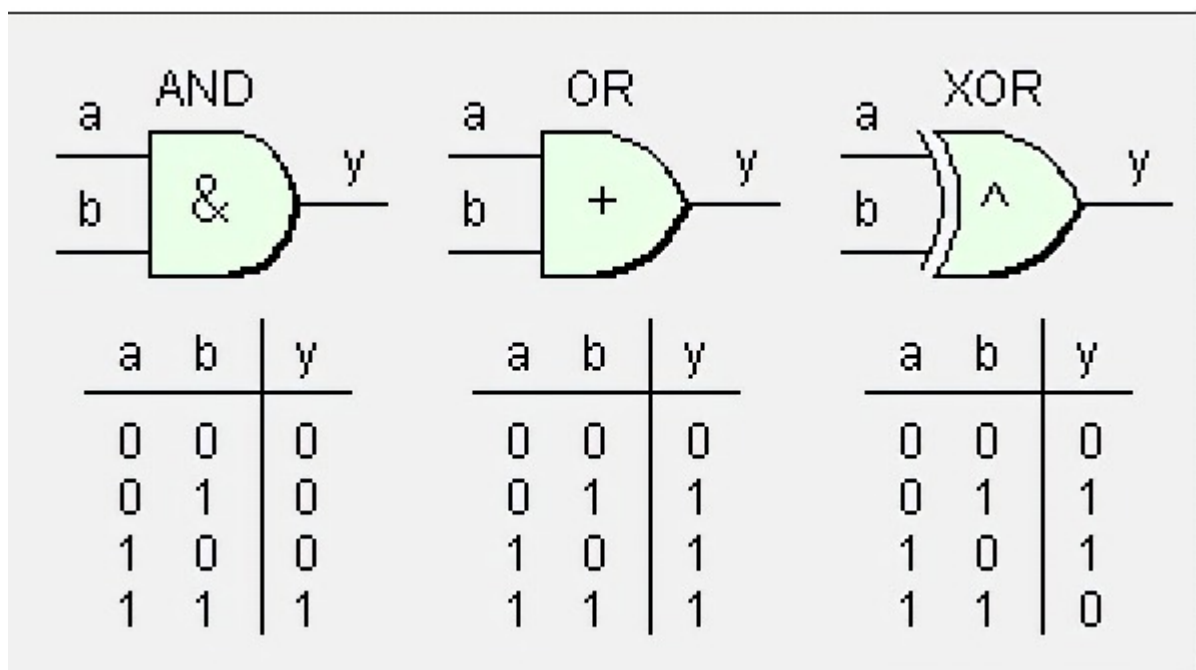
`loss = F.mse_loss ( prediction, labels)`

`loss.backward()`

`optim.step()`

`optim.zero_grad()`

Your task will be to make and train your own neural networks :) So to transform the pseudocode above to working code. Your neural networks will emulate the 'and', 'or' and 'xor' functions:



As can be seen in the image above, there are only 4 possible inputs to the neural network, [0,0], [0,1],[1,0] and [1,1] and 2 possible outputs 0 or 1.

For each of these functions, first try to make a single layer network (logistic regression model). These models can only solve linear problems. If a single layer networks fails, try to make a two layer neural network, which can solve non-linear problems. You can experiment with activation functions and number of neurons, but try to keep it simple :)

Have fun!

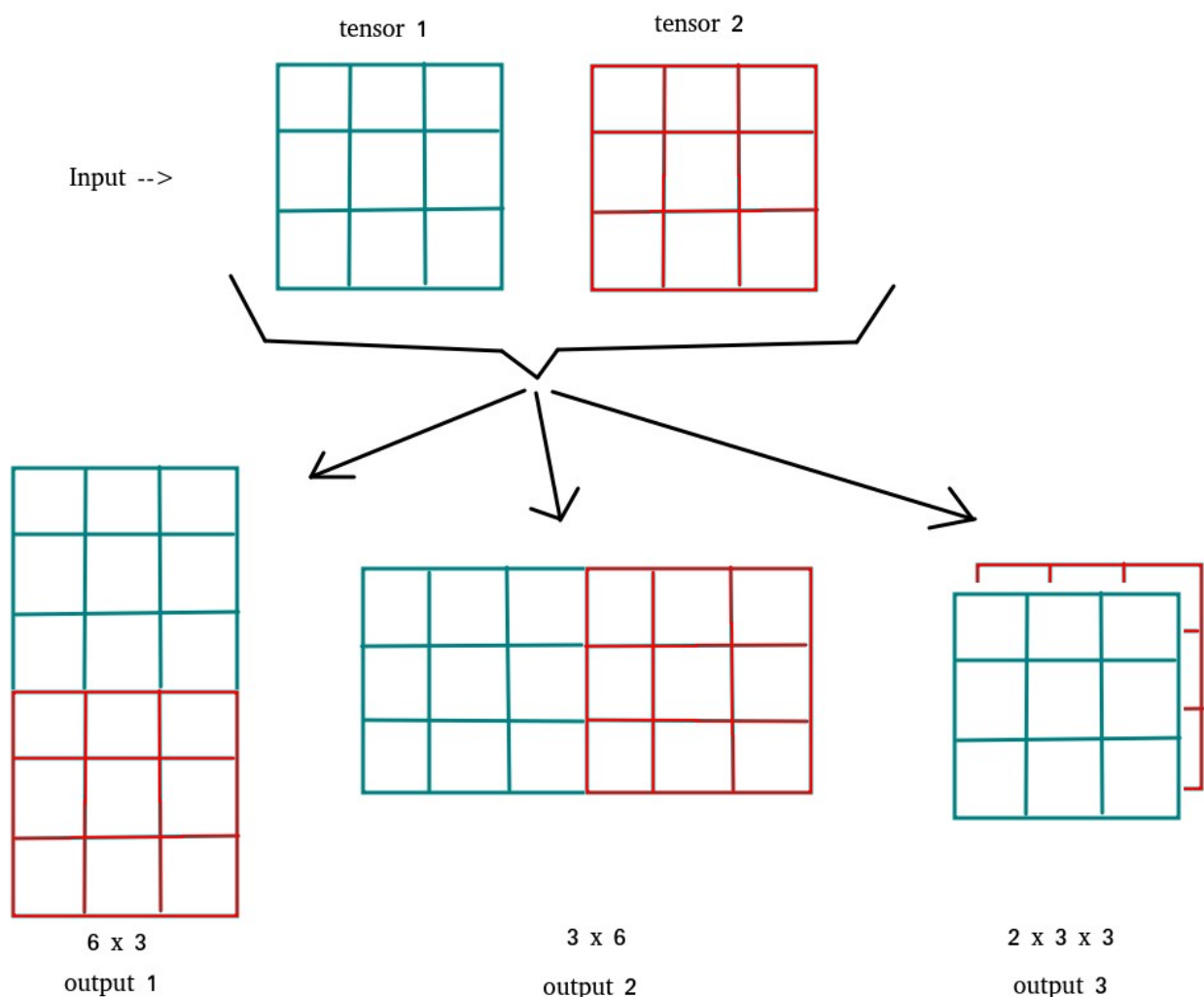
(p.s. make sure the data and labels are float tensors...)

---

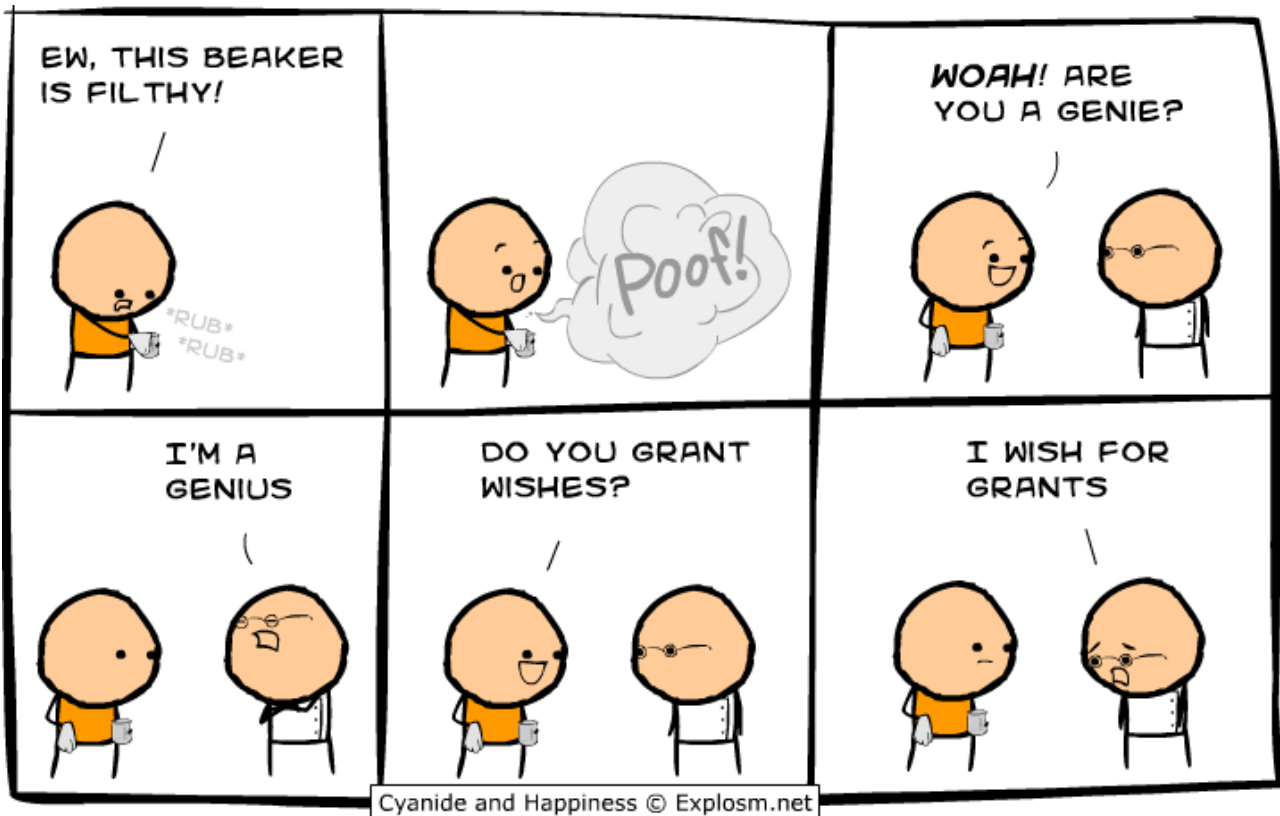
### Homework question 5.

The concatenation function of pytorch can be quite useful. Find out how this functions is used with the pytorch library.

Given two matrices with the same size (3,3) try to concatenate them on both dimensions but also on a new third dimension:



And lastly here is a fun science comic :)



If you think something is not explained well enough or something is missing, put it in the machine learning channel on slack, and I will include it next time!